Inter-Office Memorandum

To	Mesa Users	Date	October 27, 1980
From	Ed Satterthwaite	Location	Palo Alto
Subject	**Mesa 6.0 Compiler Update**	Organization	PARC/CSL

XEROX

Filed on: [Iris]<Mesa>Doc>Compiler60c.bravo, Compiler60d.bravo, and Compiler60.press

This memo describes changes to the Mesa language and compiler that have been made since the release of Mesa 5.0 (April 9, 1979).

Definitions of syntactic phrase classes used but not defined in this document come from the *Mesa Language Manual, Version 5.0,* Appendix F.

Compatibility

Because of changes in symbol table and BCD formats, you must recompile all existing Mesa programs after obtaining recompiled versions of the interfaces and packages that they depend upon.

Language Changes

Our goal for language compatibility has been to accept any valid Mesa 5 source program as a valid Mesa 6 source program. We are aware of the following incompatibilities:

There are some new reserved words, as follows:

FREE PRED PROC REJECT SEQUENCE SUCC UNCOUNTED ZONE

Some of the quoted file names that appear in DIRECTORY clauses have different interpretations. Text inside angle brackets is no longer ignored; it is treated as the name of a local subdirectory (but we do not recommend using local subdirectories for Mesa programs at the present time).

The orders of evaluating the items in constructors (including argument lists) and the operands of infix operators (except AND and OR) have changed somewhat. In particular, programs that assumed a left-to-right order of procedure calls in these contexts (e.g., *Divide*[*Pop*[], *Pop*[]]) are unlikely to work correctly.

The assignment operation is no longer available for updating objects containing MONITORLOCK or CONDITION values; updating of such objects must be done component-by-component.

The granularity of packed arrays has changed. If the components of a packed array can be stored in four or less bits, the storage structures defined by the declaration of that array will differ between Mesa 5 and Mesa 6. This is a potential problem in reading files created by earlier versions of Mesa. Also, the DESCRIPTOR operator cannot be applied to packed arrays occupying less than a word.

If you have been using type REAL, check with the supplier of your floating point package to determine the effect of Mesa 6 changes in that area.

Bug Fixes

A large number of Mesa 5 bugs have been fixed. The most notable of these involve

expansion of inline procedures that are defined in DEFINITIONS modules,

expansion of inline procedures when an argument is itself an expanded inline,

proper retention of the tag of a variant record within an arm of a discriminating selection,

proper identification of object files that have been renamed.

Because of certain bug fixes, the compiler may reject previously acceptable programs or may issue new warning messages.

As usual, the list of compiler-related change requests closed by Mesa 6.0 will appear separately as part of the Software Release Description.

Language Rationalization

Mesa 6 attempts to remove certain minor anomalies and to add some obvious generalizations to the existing language.

Syntactic Glitches Removed

List Punctuation: For most lists that are explicitly bracketed by symbols other than [and], the allowable forms are described by the following meta-BNF:

```
list  ::=        empty | item | item separator list
```

In other words, the list may be empty or may be a sequence of items separated by, and optionally terminated by, a separator. This rule now applies to the following constructs:

form	*separator*	*notes*
VariantList	,	
CatchSeries	;	ANY must come last
ChoiceSeries	;	
ExitSeries	;	
StatementSeries	;	
StmtChoiceSeries	;	
ChoiceList	,	
ExprChoiceList	,	

When the bracketing symbols are [and] or when the length of a list is significant, a trailing separator is not allowed unless it is semantically meaningful (as in constructors and extractors), but empty lists are allowed. This change affects the following syntactic entities:

UsingClause: The form USING [ ] is now permitted (e.g., to emphasize that an interface is only being exported).

VariantFieldList: In the declaration of a variant record, the form [ ] (empty brackets) may be used instead of NULL (and is recommended).

FieldList: The form [ ] is permitted in the declaration of a ParameterList or ReturnsClause (it is equivalent to omitting the list or clause).

Directory: The form DIRECTORY ; may be used in place of an empty directory.

ImportsList (ExportsList): The form IMPORTS (EXPORTS) may be used when the corresponding list is empty.

The last two items reflect the view that DIRECTORY, IMPORTS and EXPORTS introduce formal parameter lists, even though their punctuation omits [and].

Declarations in Loop Bodies: A declaration series can now appear between DO and ENDLOOP (after the OPEN and ENABLE clauses, if any); no additional bracketing is required. Its scope is limited to the DeclarationSeries and StatementSeries in the loop body.

A Bit Less Verbiage

PROC: PROC is accepted as a short form of PROCEDURE.

Statement Brackets: The bracket pair { } can be used any place the bracket pair BEGIN END can be used (but not conversely).

Semantic Glitches Removed

LONG *Arithmetic*: Compile-time evaluation of expressions with constant operands now works for 32-bit quantities just as it does for 16-bit quantities.

Dereferencing with OPEN: A pointer expression following OPEN or WITH will be dereferenced an arbitrary number of times (not just once) to obtain an expression designating a record.

Renaming with OPEN: An OPEN clause may give an alternative local name to an interface that it opens (formerly, only to a record).

Nested Extractors: An ExtractItem may itself be an Extractor. This allows extraction from records embedded within records. In particular, this form is useful in situations where a single-component record is not automatically converted to its single component, e.g., extraction from a record that is the only component of a return record.

Extractor Expressions: An assignment with an Extractor as its left side may be used as an expression. This allows, among other things, multiple extraction. The value of such an assignment is the value of its right side.

Interface Aliases: An interface module may have multiple identifiers (preceding ": DEFINITIONS"). In the DIRECTORY clause of another module, you can reference that interface using any one of the identifiers. A type obtained from such an interface is equivalent to the same type obtained through a naming path that uses any other identifier of the interface.

New Language Features

Extended Defaults

You can associate a default initial value with a type (not just with a field of a record). If a type is constructed from other types using one of Mesa's type operators (e.g., RECORD), the default value for that type is determined by the default values of the component types and by rules associated with each operator. When you declare a named type, you have the option of explicitly specifying a default for that type.

With this extension, you will find that uses of defaults in Mesa generally fall into two classes. Default values for fields of records make the corresponding constructors more concise and more convenient to use. On the other hand, the usual reason for associating a default initial value with a type is to ensure that storage allocated for that type is well-formed, i.e., that any variable of such a type always has a meaningful value. There is some interaction between these uses; the default value of a record type is partly determined by any default values specified for its fields, and a record field may inherit its default value from the type of that field. The details appear below.

The rules for inheritance of defaults are designed to provide the following property (currently not quite preserved by sequence or variant record types): if a type *T* has been given a non-NULL default value, any type derived from *T* will have a defined and non-NULL default value for any embedded component of type *T*. Because of the potential cascading effect implied by this, you should carefully consider the relative costs and benefits of specifying a default, especially one that does not include NULL as an alternative.

Defaults are ignored in determining equivalence and conformance of types. Thus it is possible to have two compatible types with different default initializations.

Specification of Default Initialization

None of the built-in types (INTEGER, CARDINAL, BOOLEAN, CHARACTER, STRING and REAL) has a default initial value.

The following rules determine the default initial value of a type designated by an expression involving a type operator:

> The default initial value for a type constructed using RECORD (or ARRAY) is defined field-by-field (or element-by-element). For each field (element), it is the default value for that field if there is one; otherwise, it is the default initial value for the type of that field (element) or is undefined if there is no such default.
>
> The default initial value for any port type, constructed using PORT, is NIL (see below).
>
> Types constructed using other operators have no implied default initialization.

The default initial value of a type designated by a declared type identifier *T* depends upon the form of the declaration of *T*, as follows:

> *T:* TYPE = *TypeExpr*;
>
> *T* receives all the attributes of *TypeExpr* including any default.
>
> *T:* TYPE = *TypeExpr* _ *e*;
>
> *T* receives all the attributes of *TypeExpr* except that its default initial value is *e*.

Examples

```
Flag: TYPE = BOOLEAN _ FALSE;
Rec1: TYPE = RECORD [f: Flag];              -- default value is [f: FALSE]
Rec2: TYPE = RECORD [f: Flag] _ [ ];        -- ditto (the field defaults)
Rec3: TYPE = RECORD [f: Flag] _ [TRUE];     -- explicit default
Rec4: TYPE = Rec3;                          -- default value is [f: TRUE]
Rec5: TYPE = Rec3 _ [f: FALSE];             -- default value is [f: FALSE]
```

Any DefaultSpecification is acceptable in a type declaration (see *Mesa Language Manual, Version 5.0*, page 37). A declaration giving a type *T* a NULL default cannot, however, equate *T* to a type with a default that does not include NULL. A default appearing in a type definition within a DEFINITIONS module must be either NULL or an expression with a compile-time constant value.

Default values associated with types are used

to initialize local variables of procedures and programs, in the absence of explicit initialization,

to initialize variables that are dynamically allocated using NEW, in the absence of explicit initialization (see below),

to construct records (except argument and result records), in the absence of an explicit value for a field in the constructor and of a default value for that field in the record declaration,

to construct arrays, in the absence of an explicit value for an element (see below).

Defaults in Argument and Result Records

You may specify default values for the fields of argument and result records. Such default values must be constructed from constants or variables that are declared outside of the procedure type definition. In particular, you cannot use a value of another field of the same record or, in the case of a result record, a value from the associated argument record to define such a default.

You may omit a field in the constructor of an argument or result record only if the definition of that record specifies an explicit default value for the field; default initial values associated with the *types* of such fields are *not* inherited (for example, this protects you from assigning a value to a return variable and then forgetting to mention it in a RETURN statement, causing the default for its type to be returned). On the other hand, protection against ill-formed storage *is* inherited; you may not void or elide a field unless the type of that field allows a NULL initialization.

Any defaults that you specify in the declaration of a result record serve two purposes. Since the fields of such a record can be used as local variables within the procedure body, a default specification affects the initialization of those variables; in addition, it allows abbreviation in the constructors of the corresponding return records. The precise rules are the following:

Upon entry to a procedure, each field of the result record is initialized with the default value specified for that field, if any; otherwise, with the default initial value for the type of that field, if there is one; otherwise, its initial value is undefined.

If a RETURN is followed by an explicit constructor, the default specifications appearing in the declaration of the result record control the values of any omitted or elided fields, *even if other assignments have been made to the result variables within the procedure body*. If the RETURN stands by itself, without such a constructor, or if the RETURN is implicit, the return record is constructed using the current values of the result variables.

Examples

```
T: TYPE = INTEGER _ 1;

Proc1: PROC [i: INTEGER _ 0, j: T];

Proc2: PROC RETURNS [m: T, n: INTEGER _ 2] = {
  -- m initialized to 1 (from T), n to 2
  Proc1[j: 3];                 -- Proc1[i:0, j:3];
  Proc1[i: 3];                 -- illegal (j does not default to 1)
  ...
  m _ 4;  n _ 5;
  ...  RETURN;                 -- returns [4, 5]
  ...  RETURN [m, n];          -- also returns [4, 5]
  ...  RETURN [m];             -- returns [4, 2]
  ...  RETURN [NULL, n];       -- illegal (declaration of T disallows voiding of m)
  ...  RETURN [, n];                     -- ditto (m does not default to 1 or 4)
  ...  RETURN [6, 7];          -- returns [6, 7]
  ... };                       -- implicitly returns [4, 5]
```

Defaults and Variant Records

You may specify a default for the entire variant part in the declaration of a variant record type. In the absence of such a specification, the default value of that part, including the tag, is undefined with respect to the undiscriminated record type.

The default initial value of a discriminated variant record type has a tag value corresponding to the discriminating adjective, and defaults for the other fields of the variant part are those implied by the fields selected by that tag. In particular, the declaration or allocation of a variable with discriminated record type sets the tag correctly.

Example

```
VRec: TYPE = RECORD [
  common: INTEGER _ 0,
  variant: SELECT tag: * FROM
    red => [r1: BOOLEAN _ FALSE],
    green => [g1: INTEGER _ 0]
    ENDCASE _ red[TRUE] | NULL];

v: VRec;                    -- initial value is [common: 0, variant: red[r1: TRUE]]
v1: VRec _ [common: 10];    -- initial value is [common: 10, variant: red[r1: TRUE]]
v2: VRec _ [variant: NULL]; -- tag and variant part are undefined, v2.common = 0
v3: VRec _ NULL;            -- illegal (declaration of common does not allow NULL)
rv: red VRec;               -- initial value is [common: 0, variant: red[r1: FALSE]]
gv: green VRec;             -- initial value is [common: 0, variant: green[g1: 0]]
```

Defaulted Array Elements

Elements in an array constructor may be voided or elided. Omission of elements is permitted in a keyword constructor (see below) but not in a positional constructor. The empty constructor ([ ]) is a keyword constructor with all items omitted. An elided or omitted element receives the default value for the type of the components of the array (if any); the value of a voided element is undefined.

ALL abbreviates a positional constructor of the appropriate length; thus ALL[ ] elides all elements (defaulting if possible) and ALL[NULL] voids all positions.

Keyword Array Constructors

You can use keyword array constructors when the index type of the array is an enumeration or subrange thereof. The acceptable keywords are the constants appearing in the enumeration. In the case of a subrange, the endpoints must be defined by expressions involving only those constants, the operators FIRST, LAST, SUCC and PRED, and identifiers equated by declaration to such expressions. If the component type of the array has a defined default value (including NULL), keyword items can be omitted; the corresponding elements receive the default value.

Packed Arrays

If you specify the PACKED attribute for an array type, the granularity of packing is 1, 2, 4, 8 or 16*n* bits and is determined by the component type of the array (formerly just 8 or 16*n* bits).

The value of the construct SIZE[*T*, *n*] is the size, in words, of the storage required by a packed array of *n* items of type *T*. (SIZE[*T*] continues to yield the number of words occupied by a single item of type *T*.)

Example

```
    Bit: TYPE = BOOLEAN _ FALSE;
    BitSet: TYPE = PACKED ARRAY Color OF Bit;
    AllBits: BitSet = ALL[TRUE];
    threeBits: BitSet _ [yellow: TRUE, red: TRUE, blue: TRUE];
```

Successor and Predecessor Operations

The operators SUCC and PRED operate upon values of any ordered type except REAL. For numeric and character types, SUCC[*x*] and PRED[*x*] are equivalent to *x*+1 and *x* 1 respectively. For enumerated types, the values are the successor and predecessor of *x* in the enumeration; a bounds fault occurs if there is no such element *and* you requested bounds checking.

Directories

You can now override the association established by the DIRECTORY clause between the names of included modules and the names of the files containing those modules. Any file names implied by convention can be omitted entirely; they will be computed from the interface identifiers.

Syntax

```
    IncludeList          ::=     IncludeItem
                         |       IncludeList , IncludeItem

    IncludeItem          ::=     identifier UsingClause
                         |       identifier : FROM FileName UsingClause
                         |       identifier : TYPE UsingClause
                         |       identifier : TYPE identifier UsingClause

    UsingClause          ::=     empty | USING [ IdList ] | USING [ ]
```

The (initial) identifier in an IncludeItem names a module. If you specify the name of the file containing that module when you invoke the compiler (as a keyword parameter with keyword identifier, see below), that name is used, even if a FileName appears in the IncludeItem. Otherwise, if such a FileName appears, it is used. If you supply neither a compile-time argument nor a FileName, the name `identifier.bcd` is used.

One approach to describing systems built from collections of Mesa modules views the DIRECTORY clause as the declaration of (compile-time) formal parameters of type TYPE. Mesa 6 provides the final two forms of IncludeItem primarily for compatibility with this view. The identifier preceding the colon names the formal parameter; it is also used to derive a file name as described above. The identifier following TYPE constrains the set of acceptable actual parameters; it must match the ModuleName used in the definition of the module that you intend to include (see the *Mesa Language Manual, Version 5.0,* Section 7.2).

You can use the final form to change the name by which one module is known within another, notably to avoid duplicate names in a DIRECTORY clause. For example, you might need to reference two different versions or parameterizations of an interface *Defs* within a single program. The IncludeItems

> *LongDefs:* TYPE *Defs,*
> *ShortDefs:* TYPE *Defs*

declare *LongDefs* and *ShortDefs* as identifiers within that program of possibly different modules, each with the ModuleId *Defs*. As IncludeItems, the forms

> *identifier*
> *identifier:* TYPE

each abbreviate

> *identifer:* TYPE *identifier*

and the name in the DIRECTORY must be identical to the ModuleId if you use one of these forms.

Implicitly Imported Interfaces

An implicitly imported interface is one from which imported values are required for binding the free variables of another, explicitly imported interface. For example, interface *D1* might import interface *D2* to gain access to a procedure or exported variable supplied by the latter. *D2* is then implicitly imported by any program module *M* that imports *D1* (see the *Mesa Language Manual, Version 5.0*, Section 7.4.4).

In Mesa 6, the free variables of *D2* that are used by *D1* are bound to the *principal instance* of *D2* in *M*. An import is a principal instance if it is the only instance of that interface imported by a module *or* if it is unnamed. Furthermore, if *M* imports no instances of *D2*, a principal instance will be created automatically. If module *M* has no other reason to mention *D2*, *D2* then need not appear in either the DIRECTORY or the IMPORTS list of *M*. Explicitly importing a principal instance of *D2* in such a situation is not an error, and you must do so if

> you plan to use positional notation to specify the imports of *M* in a C/Mesa configuration description, since the positions of automatically created interface instances are not defined, or
>
> you already import more than one instance of *D2*, each of which is named.

In a C/Mesa configuration, principal instances of interfaces are *not* supplied automatically; you must import them explicitly if they cross (sub)configuration boundaries.

Real Numbers

Mesa 6 has adopted the proposed IEEE standard for floating-point arithmetic (see, e.g., Coonen, An implementation guide to a proposed standard for floating-point arithmetic, *Computer*, January 1980, pp. 68-79). In support of this, the language provides floating-point literals and the compiler performs a limited number of operations upon floating-point constants.

Syntax

primary	::=	... | realLiteral
realLiteral	::=	unscaledReal
	|	unscaledReal scaleFactor
	|	wholeNumber scaleFactor
unscaledReal	::=	wholeNumber fraction
	|	fraction
fraction	::=	. wholeNumber
scaleFactor	::=	E optSign wholeNumber | e optSign wholeNumber
optSign	::=	empty | + |
wholeNumber	::=	digit | wholeNumber digit

An unscaledReal has its usual interpretation as a decimal number. The scaleFactor, if present, indicates the power of 10 by which the unscaledReal or wholeNumber is to be multiplied to obtain the value of the literal.

Mesa 6 represents REAL numbers by 32 bit approximations as defined in the IEEE standard. The rounding mode used to convert literals is "round-to-nearest." A literal that overflows the internal representation is an error; one that underflows is replaced by its so-called "denormalized" approximation. In Mesa 6, the value of the unscaledReal in a literal must be a valid LONG INTEGER when the decimal point is deleted.

No spaces are allowed within a realLiteral. Note that such a literal can begin, but not end, with a decimal point. Thus the interpretation of [0...1) is unambiguous (but perhaps surprising; use [0 .. .1) or [0.0..0.1) instead).

Operations

The compiler performs the following operations involving floating-point constants:

Unary negation (with 0 = 0)
ABS
Fixed-to-Float (in "round-to-nearest" mode).

Other operations are deferred until runtime, even if all their operands are constant, so that the programmer can control the treatment of rounding and exceptions (see the proposed standard).

Unless you specify the compilation option -f (see below), the compiler generates instructions for floating-point operations that require hardware or microcode support. If you are in doubt about the state of your machine or its microcode, see a local floating-point expert.

Machine Dependent Enumerations

Sometimes a programmer can enumerate the values of some type but requires control of the encoding of each value or of the number of bits used to represent the type (usually for future expansion). Mesa 6 provides machine-dependent enumerations for such applications.

Syntax

EnumerationTC	::=	MachineDependent { ElementList }
MachineDependent	::=	empty | MACHINE DEPENDENT
ElementList	::=	Element | ElementList , Element
Element	::=	identifier
	|	identifier (Expression)
	|	(Expression)

Examples

Status: TYPE = MACHINE DEPENDENT {*off*(0), *ready*(1), *busy*(2), *finished*(4), *broken*(7)}

Color: TYPE = MACHINE DEPENDENT {*red*, *blue*, *green*, (255)} -- reserve 8 bits

Each Expression in an EnumerationTC must denote a compile-time constant, the value of which is an unsigned integer.

In an enumerated type with the MACHINE DEPENDENT attribute, the values used to represent the enumeration constants are assigned according to the following rules. If a parenthesized expression follows the element identifier, the value of that expression is used; otherwise, the representation of an element is one greater than the representation of the preceding element. If you specify only a representation, the corresponding element (normally a place holder) is anonymous. If the representation of the initial element is not given, the value zero is used.

You cannot explicitly specify the representation of any element unless the attribute MACHINE DEPENDENT appears in the type constructor. Two element identifiers cannot be represented by the same value (either given explicitly or determined implicitly as described above). The ordering of elements determined by position in the ElementList must agree with the ordering determined by the (unsigned) arithmetic ordering of the representations.

Sparse Enumerations

A machine-dependent enumerated type is *sparse* if there are gaps within the set of values used to represent the constants of that type or if the smallest such value is not zero. Mesa 6 takes the following position on gaps: they are filled by valid but anonymous elements of the enumerated type. These elements can be generated only by the operators FIRST, SUCC and PRED (or by the iteration forms that implicitly use these operators).

If you use a sparse enumerated type as the index type of an array, the array itself will have components for all elements of the enumeration, including the anonymous ones. The latter are awkward to access (except through ALL) and may cause problems in constructors, comparison operations, etc., as well as wasted space. (For example, ARRAY *Color* OF INTEGER would occupy 256 words.)

Machine Dependent Records

Machine-dependent records are provided for situations in which the exact position of each field is important. In Mesa 6, you can explicitly specify word- and bit-positions in the declaration of the record type. This form provides better documentation and usually is easier to use than the previous, purely positional form. You should use it in preference to the old form, which remains for compatiblity.

Syntax

```
VariantFieldList        ::=     CommonPart FieldId : Access VariantPart
                        |       VariantPart
                        |       NamedFieldList
                        |       UnnamedFieldList
                        |       empty
CommonPart              ::=     NamedFieldList , | empty
NamedFieldList          ::=     NamedField | NamedFieldList , NamedField
NamedField              ::=     FieldIdList :
                                Access TypeSpecification DefaultOption
FieldIdList             ::=     FieldId | FieldIdList , FieldId
FieldId                 ::=     identifier | identifier ( FieldPosition )
Tag                     ::=     FieldId | ...
FieldPosition           ::=     Expression : Expression .. Expression
                        |       Expression
```

Examples

```
InterruptWord: TYPE = MACHINE DEPENDENT RECORD [
   channel (0: 8..10): [0..nChannels),          -- nChannels <= 8
   device (0: 0..7): DeviceNumber,
   stopCode (0: 11..15): MACHINE DEPENDENT {finishedOK(0), errorStop(1), powerOff(3)},
   command (1: 0..31): ChannelCommand];

Node: TYPE = MACHINE DEPENDENT RECORD [
   type (0: 0..15): TypeIndex,
   rator (1: 0..13): OpName,
   rands (1: 14..47): SELECT valence (1: 14..15): * FROM
      nonary => [],
      unary => [left (1: 16..31): POINTER TO Node],
      binary => [left (1: 16..31), right (1: 32..47): POINTER TO Node]
      ENDCASE]
```

An identifier with an explicitly specified FieldPosition can occur only in the declaration of a field of a record defined to have the MACHINE DEPENDENT attribute. If the position of any field of a record is specified, the positions of all must be. Each Expression in a FieldPosition must denote a compile-time constant, the value of which is an unsigned integer.

The first expression appearing in a FieldPosition specifies the (zero-origin) record-relative index of the word containing the start of the field; the second and third specify the indices (zero-origin) of the first and last bits of the field with respect to that word. The second and third expressions may specify a bit offset greater than the word size if the word offset is adjusted accordingly. Similarly, the difference between the second and third expressions may exceed the word size. If the bit positions are not specified, a specification of 0..*n***WordSize*-1 is assumed, where *n* is the minimum number of words required by the type of the field.

Each field must be at least wide enough to store any value of the corresponding type. Values are stored right-justified within the fields. The current implementation of Mesa imposes the following additional restrictions on the sizes and alignment of fields:

> A field smaller than a word (16 bits) cannot cross a word boundary.
>
> Any field occupying a word or more must begin at bit zero of a word and have a size that is a multiple of the word size.
>
> A variant part may begin at any bit position (as determined by its tag field).
>
> If the sizes of all variants of a record type are less then a word, those sizes must be equal; otherwise, the size of each variant of the type must be a multiple of the word length.

In the definition of a machine-dependent record type, explicitly specified field positions must not overlap. For a variant record type, this requirement applies to the variant part (including the tag) considered in conjunction with the fields of the common part; the tag and fields particular to each variant must lie entirely within the variant part.

The order of fields in a record type declaration need not agree with the order of those fields in the representation of the record; however, no gaps are permitted. For variant records, the fields of at least one variant (including the tag field) must fill the position specified for the variant part.

Dynamic Storage Allocation

In Mesa 6, you can use special constructs to describe the dynamic allocation and deallocation of variables. You are still responsible for managing the storage and guarding against dangling pointers; the new features handle certain routine aspects of allocation and deallocation (such as computing sizes), provide proper default initialization of newly allocated variables, and reduce the total number of LOOPHOLEs required to deal with an allocator.

Zones

Allocation and deallocation are done with respect to *zones*. A zone need not be associated with any specific storage area; it is just an object characterized by procedures for allocation and deallocation as described below. The storage managed by a zone in Mesa 6 is said to be *uncounted*. In such zones, object management is the responsibility of the programmer, who must explicitly program the deallocation.

To use an uncounted zone, you must provide the procedures that manage the zone and implement the required set of operations. Many users will be able to import a suitable implementation from a standard package; the details of writing such packages are discussed below.

A zone object has a value and a type. You will normally obtain a zone value by calling a procedure exported by some package implementing zones. Typically, such a procedure constructs a zone (and perhaps an initial storage pool) according to user-supplied parameters.

The type of a zone value must belong to a new class of types, called zone types. Mesa 6 provides two such types, UNCOUNTED ZONE and *MDSZone*. Transactions with objects having these types are generally in terms of LONG POINTER and POINTER values respectively (see below).

Syntactically, UNCOUNTED ZONE is a type constructor. *MDSZone* is a predeclared identifier; you may think of it as a synonym for *MDS* RELATIVE UNCOUNTED ZONE (which you currently cannot write directly).

You may declare variables having zone types (for which fixed initialization is recommended). Zone types may also be used to construct other types. In particular, you may choose to deal with pointers to zones; the NEW and FREE constructs described below provide automatic dereferencing.

Allocating Storage

The operator NEW allocates new storage of a specified type, initializes it appropriately, and returns a pointer to that storage. The NEW operation is considered an attribute of a zone, which must be specified explicitly.

Syntax

```
    Primary          ::=        ...
                     |          Variable . NEW [ TypeSpecification Initialization OptCatch ]
                     |          ( Expression ) . NEW [ TypeSpecification Initialization OptCatch ]
    Initialization   ::=        empty | _ InitExpr | = InitExpr
    OptCatch         ::=        empty | ! CatchSeries
```

The value of the Variable or Expression identifies the zone to be used, either directly or after an arbitrary number of dereferencing operations. The TypeSpecification determines the type of the allocated object. If an InitExpr is provided, it must conform to the specified type and its value is used to initialize the new object; otherwise, the default value associated with that type (if any) is used. Only signals raised or propagated by the allocation procedure activate a CatchSeries attached to NEW.

The value of the Primary is a pointer to the newly allocated object. The type of that pointer depends upon the type of the zone and the form of the Initialization. If the argument of NEW is some type *T*, the type of the result is

LONG POINTER TO *T*, if the type of the zone is equivalent to UNCOUNTED ZONE

POINTER TO *T*, if the type of the zone is equivalent to *MDSZone*.

If you specify fixed (=) initialization, the result is a read-only pointer with type LONG POINTER TO READONLY *T* or POINTER TO READONLY *T* respectively.

The InitExpr cannot be the special form for string body initialization ([Expression]). You can, however, allocate string bodies with dynamically computed sizes by using a new form of TypeSpecification (see below). If you do so, the Initialization must be empty.

Releasing Storage

Uncounted zones have FREE operations. When applied to an object, this operation releases the storage allocated for that object.

Syntax

```
    Statement        ::=        ...
                     |          Variable . FREE [ Expression OptCatch ]
                     |          ( Expression ) . FREE [ Expression OptCatch ]
```

The zone used in a FREE operation is determined as described for NEW; it should be the zone from which the variable was originally allocated. The argument of FREE is the *address* of a pointer to the variable to be deallocated; FREE sets the pointer to NIL and deallocates the storage for the variable.

Only signals raised or propagated by the deallocation procedure activate a CatchSeries on a FREE.

Implementing Uncounted Zones

This section describes the assumptions currently made by the compiler about the user-supplied implementations of uncounted zones. These assumptions are compatible with the style of "object-oriented" programming that has proven successful in a number of applications. You need to read this section only if you are designing the interface between a storage management package and the zone features of the language.

An uncounted zone dealing with LONG POINTER values is represented by a two word value, which the compiler assumes to be a long pointer compatible with the following skeletal structure:

```
UncountedZoneRep: TYPE = LONG POINTER TO MACHINE DEPENDENT RECORD [
   procs (0:0..31): LONG POINTER TO MACHINE DEPENDENT RECORD [
      alloc (0): PROC [zone: UncountedZoneRep, size: CARDINAL] RETURNS [LONG POINTER],
      dealloc (1): PROC [zone: UncountedZoneRep, object: LONG   POINTER]
      -- possibly followed by other fields-- ],
   data (2:0..31): LONG POINTER                          -- optional, see below
   -- possibly followed by other fields-- ];
```

If *z* is an uncounted zone, the code generated for *p* _ *z*.NEW[*T*] is equivalent to

```
p _ z^.procs^.alloc[z, SIZE[T]]
```

and the code generated by *z*.FREE[@*p*] is equivalent to

```
{temp: LONG POINTER _ p; p _ NIL; z^.procs^.dealloc[z, temp]}.
```

Within this framework, you may design a representation of zone objects appropriate for your storage manager. In general, you should create an instance of a *finger* (the record with fields *procs* and *data*) for each instance of a zone. The record designated by the *procs* pointer can be shared by all zones with the same implementation. The *data* pointer normally designates a particular zone and/or the state information characterizing that zone. Note that the compiler makes no assumptions about the designated object and does not generate any code referencing the *data* field. The extra level of indirection provided by that field is not obligatory; you may replace it with state information contained directly in the finger (but following the *procs* field).

The compiler assumes a similar (but single word) representation for an *MDSZone* value; the skeletal structure is as follows:

```
MDSZoneRep: TYPE = POINTER TO MACHINE DEPENDENT RECORD [
   procs (0:0..15): POINTER TO MACHINE DEPENDENT RECORD [
      alloc (0): PROC [zone: MDSZoneRep, size: CARDINAL] RETURNS [POINTER],
      dealloc (1): PROC [zone: MDSZoneRep, object: POINTER]
      -- possibly followed by other fields-- ],
   data (1:0..15): POINTER                  -- optional
   -- possibly followed by other fields-- ];
```

Sequences

A *sequence* in Mesa is an indexable collection of objects, all of which have the same type. In this respect, a sequence resembles an array; however, you need not specify the length of the sequence when its type is declared, only when an instance of that type is created. Mesa 6 provides sequence-containing types for applications in which the size of a dynamically created array cannot be computed statically. Note, however, that only a subset of a more general design for sequences has been implemented. The contexts in which sequence types may appear are somewhat restricted, as are the available operations on them. We believe that the subset provides enough functionality to accomodate most uses of sequences, but you will encounter a number of annoying and sometimes inconvenient restrictions that you must take note of in your Mesa 6 programming.

One can view a sequence type as a union of some number of array types, just as the variant part of a variant record type can be viewed as a union of some (enumerated) collection of record types. Mesa adopts this view, particularly with respect to the declaration of sequence-containing types, with the following consequences:

> A sequence type can be used only to declare a field of a record. At most one such field may appear within a record, and it must occur last.
>
> A sequence-containing object has a tag field that specifies the length of that particular object and thus the set of valid indices for its elements.

To access the elements of a sequence, you use ordinary indexing operations; no discrimination is required. In this sense, all sequences are overlaid, but simple bounds checking is sufficient to validate each access.

Uses of sequence-containing variables must follow a more restrictive discipline than is currently enforced for variant records. The (maximum) length of a sequence is fixed when the object containing that sequence is created, and it cannot subsequently be changed. In addition, Mesa 6 imposes the following restrictions on the uses of sequences:

> You cannot embed a sequence-containing record within another data structure. You must allocate such records dynamically and reference them through pointers. (The NEW operation has been extended to make allocation convenient.)
>
> You cannot derive a new type from a sequence-containing type by fixing the (maximum) length; i.e., there is no analog of a discriminated variant record type.
>
> There are no constructors for sequence-valued components of records, nor are such components initialized automatically.

The following sections describe sequences in more detail.

Defining Sequence Types

You may use sequence types only to declare fields of records. A record may have at most one such field, and that field must be declared as the final component of the record:

Syntax

```
    VariantPart         ::=     . . .
                        |       PackingOption SEQUENCE SeqTag OF TypeSpecification
```

```
SeqTag            ::=    identifier : Access  BoundsType
                  |      COMPUTED BoundsType

BoundsType        ::=    IndexType


TypeSpecification ::=    . . .
                  |      TypeIdentifier [ Expression ]
```

The TypeSpecification in VariantPart establishes the type of the sequence elements. The BoundsType appearing in the SeqTag determines the type of the indices used to select from those elements. It is also the type of a tag value that is associated with each particular sequence object to encode the length of that object. For any such object, all valid indices are smaller than the value of the tag. If *T* is the BoundsType, the sequence type is effectively a union of array types with the index types

T[FIRST[*T*] .. FIRST[*T*]), *T*[FIRST[*T*] .. SUCC[FIRST[*T*]]), ... *T*[FIRST[*T*] .. LAST[*T*])

and a sequence with tag value *v* has index type *T*[FIRST[*T*]..*v*). Note that the smallest interval in this union is empty.

If you use the first form of SeqTag, the value of the tag is stored with the sequence and is available for subscript checking. In the form using COMPUTED, no such value is stored, and no bounds checking is possible.

Examples:

```
StackRep: TYPE = RECORD [
  top: INTEGER _  1,
  item: SEQUENCE size: [0..LAST[INTEGER]] OF T]

Number: TYPE = RECORD [
  sign: {plus, minus},
  magnitude: SELECT kind: * FROM
    short => [val: [0..1000)],
    long => [val: LONG CARDINAL],
    extended => [val: SEQUENCE length: CARDINAL OF CARDINAL]
    ENDCASE]

WordSeq: TYPE = RECORD [SEQUENCE COMPUTED CARDINAL OF Word]
```

The final example illustrates the recommended method for imposing an indexable structure on raw storage.

If *S* is a type containing a sequence field, and *n* is an expression with a type conforming to CARDINAL, both *S* and *S*[*n*] are TypeSpecifications. They denote different types, however, and the valid uses of those types are different, as described below.

MACHINE DEPENDENT *Sequences*

You may declare a field with a sequence type within a MACHINE DEPENDENT record. Such a field must come last, both in the declaration and in the layout of the record, and the total length of a record with a zero-component sequence part must be a multiple of the word length. If you explicitly specify bit positions, the size of the sequence field also must describe a zero-length sequence; i.e., it must account for just the space occupied by the tag field (if any).

Examples:

> *Node*: TYPE = MACHINE DEPENDENT RECORD [
>   *info* (0: 0..7): CHARACTER,
>   *sons* (0: 8..15): SEQUENCE *nSons* (0: 8..15): [0..256) OF POINTER TO *Node*]
>
> *CharSeq*: TYPE = MACHINE DEPENDENT RECORD [
>   *length* (0): CARDINAL,
>   *char* (1): PACKED SEQUENCE COMPUTED CARDINAL OF CHARACTER]

Allocating Sequences

If *S* designates a record type with a final component that is a sequence, *S*[*n*] is a type specification describing a record with a sequence part containing exactly *n* elements. The expression *n* must have a type conforming to CARDINAL. Its value need *not* be a compile-time constant; however, you can use specifications of this form only to allocate sequence-containing objects (as arguments of NEW) or to inquire about the size of such objects (as arguments of SIZE). In particular, you cannot use *S*[*n*] to define or construct a new type or to declare a variable.

The value of the expression SIZE[*S*[*n*]] has type CARDINAL and is the number of words required to store an object of type *S* having *n* components in its sequence part.

The value of the expression *z*.NEW[*S*[*n*]] has type POINTER TO *S* (or LONG POINTER TO *S*, depending upon the type of the zone *z*). The effect of its evaluation is to allocate SIZE[*S*[*n*]] words of storage from the zone *z* and to initialize that storage as follows:

> Any fields in the common part of the record receive their default values.
>
> The sequence tag field receives the value SUCC^{n}[FIRST[*T*]], where *T* is the type of that field.
>
> The elements of the sequence part have undefined values.

To supply initial values for the fields in the common part, you may use a constructor for type *S* in the call of NEW. There are currently no constructors for sequence parts, however, and you must void the corresponding field. In any case, you must explicitly program any required initialization of the elements of the sequence part. In Mesa 6, this is true even if the element type has non-NULL default value.

Examples:

> *ps:* POINTER TO *StackRep* _ *z*.NEW[*StackRep*[100]];   -- *s.top* = 1
>
> *pn:* POINTER TO *Node* _ z.NEW[*Node*[*degree*[*c*]] _ [*info: c*, *sons:* NULL]]
>
> *pxn:* POINTER TO *extended Number* _ *z*.NEW[*extended Number*[2**k*]]

Note that *n* specifies the maximum number of elements in the sequence part and must conform to CARDINAL no matter what BoundsType T_i appears in the SeqTag. The value assigned to the tag field is SUCC^{n}[FIRST[T_i]]. A bounds fault occurs if this is not a valid value of type T_i, i.e., if $n >$ *cardinality*(T_i), *and* you have requested bounds checking.

If FIRST[T_i] = 0, SUCC^{n}[FIRST[T_i]] is just *n*, i.e., the interpretation of the tag is most intuitive if T_i is a zero-origin subrange. Usually you will specify a BoundsType (e.g., CARDINAL) with a range that comfortably exceeds the maximum expected sequence length. If, however, some maximum length *N* is important to you, you should consider using [0..*N*] as the BoundsType; then the value of the tag field in a sequence of length *n* ($n < N$) is just *n* and the valid indices are in the interval [0..*n*).

Operations on Sequences

You can use a sequence-containing type *S* only as the argument of the type constructor POINTER TO. Note that the type of *z*.NEW[*S*[*n*]] is POINTER TO *S* (not POINTER TO *S*[*n*]). If the type of an object is *S*, the operations defined upon that object are

ordinary access to fields in the common part

readonly access to the tag field (if not COMPUTED)

indexing of the sequence field

constructing a descriptor for the components of the sequence field (if not COMPUTED).

There are no other operations upon either type *S* or the sequence type embedded within *S*. In particular, you cannot assign or compare sequences or sequence-containing records (except by explicitly programming operations on the components).

Indexing: You may use indexing to select elements of the sequence-containing field of a record by using ordinary subscript notation, e.g., *s*.*seq*[*i*]. The type of the indexing expression *i* must conform to the BoundsType appearing in the declaration of the sequence field and must be less than the value of the tag, as described above. The result designates a variable with the type of the sequence elements. A bounds fault occurs if the index is out of range, the sequence is not COMPUTED, *and* you have requested bounds checking.

By convention, the indexing operation upon sequences extends to records containing sequence-valued fields. Thus you need not supply the field name in the indexing operation. Note too that both indexing and field selection provide automatic dereferencing.

Examples:

ps^.*item*[*ps*.*top*]   *ps*.*item*[*ps*.*top*]   *ps*[*ps*.*top*]   -- all equivalent

Descriptors: You may apply the DESCRIPTOR operator to the sequence field of a record; the result is a descriptor for the elements of that field. The resulting value has a descriptor type with index and component types and PACKED attribute equal to the corresponding attributes of the sequence type. By extension, DESCRIPTOR may be applied to a sequence-containing record to obtain a descriptor for the sequence part. The DESCRIPTOR operator does not automatically dereference its argument.

You cannot use the single-argument form of the DESCRIPTOR operator if the sequence is COMPUTED. The multiple-argument form remains available for constructing such descriptor values explicitly (and without type checking).

In any new programming, you should consider the following style recommendation: use sequence-containing types for allocation of arrays with dynamically computed size; use array descriptor types only for parameter passing.

Examples:

DESCRIPTOR[*pn*^]   DESCRIPTOR[*pn*.*sons*]   -- equivalent

String Bodies and TEXT

The type *StringBody* provided by previous versions of Mesa illustrates the intended properties and uses of sequences. For compatibility reasons, it has not been redefined as a sequence; the declarations of the types STRING and *StringBody* remain as follows:

```
STRING: TYPE = POINTER TO StringBody;

StringBody: TYPE = MACHINE DEPENDENT RECORD [
  length (0): CARDINAL _ 0,
  maxlength (1): --READONLY-- CARDINAL,
  text (2): PACKED ARRAY [0..0) OF CHARACTER]
```

The operations upon sequence-containing types have, however, been extended to *StringBody* so that its operational behavior is similar. In these extensions, the common part of the record consists of the field *length*, *maxlength* serves as the tag, and *text* is the collection of indexable components (packed characters). Thus *z*.NEW[*StringBody*[*n*]] creates a *StringBody* with *maxlength* = *n* and returns a STRING; if *s* is a STRING, *s*[*i*] is an indexing operation upon the text of *s*, DESCRIPTOR[*s*^] creates a DESCRIPTOR FOR PACKED ARRAY OF CHARACTER, etc.

> There are two anomalies arising from the actual declaration of *StringBody*: *s.text*[*i*] *never* uses bounds checking, and DESCRIPTOR[*s.text*] produces a descriptor for an array of length 0. Use *s*[*i*] and DESCRIPTOR[*s*^] instead.

Type TEXT

The type TEXT, which describes a structure similar to a *StringBody* as a true sequence, is predeclared in Mesa 6. Its components *length* and *maxLength* are declared to have a type compatible with either signed or unsigned numbers (but with only half the range of INTEGER or CARDINAL).

```
TEXT: TYPE = MACHINE DEPENDENT RECORD [
  length (0): [0..LAST[INTEGER]] _ 0,
  text (1): PACKED SEQUENCE maxLength (1): [0..LAST[INTEGER]] OF CHARACTER]
```

Exported Types

An *exported type* is a type designated by an identifier that is declared in an interface and subsequently bound to some *concrete type* supplied by a module exporting that interface. This is analogous to the current treatment of procedures in interfaces, where the implementations of procedures (i.e., the procedure bodies) do not appear in the interface but are defined separately. The advantages are twofold:

> The internal structure of the type is guaranteed to be invisible to clients of the interface.
>
> There are no compilation dependencies between the definition of the concrete type and the interface module. The definition of that type can be changed and/or recompiled at any time (perhaps subject to a size constraint; see below) without requiring recompilation of either the interface or any client of the interface.

The uses of an exported type are the same as those of any other type, e.g, to construct other types. The value provided by the interface is constant but has no accessible internal structure. In Mesa 6, there are two other important differences between exported procedures and exported types.

The first is a restriction necessary to ensure type safety across module boundaries. Different exporters of an interface can supply different implementations of any particular procedure in that interface. In Mesa 6, *this is not true for exported types*; all exporters of a particular type within a configuration must supply the same concrete type, which is called the *standard implementation* of that exported type. Because of this restriction, clients can safely interassign values with exported type *T*, no matter how obtained. In addition, any exporter of *T* may convert a value of type *T* to a value of the concrete type it uses to represent *T* and conversely.

The second difference is that it is not necessary to import an interface to access an exported type defined within it or to distinguish among values of such a type coming from different imported instances. This is another consequence of the fact that, in Mesa 6, all interfaces must reference the standard implementation of the exported type.

Interface Modules

An exported type is declared in an interface (DEFINITIONS) module using one of the following two forms:

> *T:* TYPE;
> *T:* TYPE [Expression];

The first of these introduces a type *T*, *no* properties of which are known in the interface or to any client of the interface. In particular, the size of *T* is not known; this is adequate (and desirable) if the interface and clients deal only with values of type POINTER TO *T*.

The second form specifies the size of the values used in the representation of the type. The value of Expression, which must denote a compile-time constant with an unsigned integer value, gives this size in units of machine words. Supplying the size of an exported type is a shorthand for exporting a set of fundamental operations (creation, _ , =, and #) upon that type. In Mesa 6, the eventual concrete type must supply the *standard implementations* of these operations, which are defined as follows:

creation	allocate the specified number of words, with no initialization
_	copy an uninterpreted bit string
=, #	compare uninterpreted bit strings

Note that a type with non-NULL default value does not have the standard creation operation. Such types cannot be exported with known size. You should therefore consider writing your interfaces in terms of POINTER TO *T*, where *T* is a completely opaque exported type and not subject to these restrictions.

Client Modules

A client has no knowledge of the type *T* beyond those properties specified in the interface. If the size is not specified there, no operations on *T* are permitted. If the size is available from the interface, SIZE[*T*] is legal; also declaration of variables (including record fields and array components) and the operations _, =, # are defined for type *T*.

Implementation Modules

An implementor exports a type *T* to some interface *Defs* by declaring the type with the required identifier, the PUBLIC attribute, and a value that is the concrete type; e.g., in

> *T:* PUBLIC TYPE = *S*;

S specifies the concrete type. If the size of *T* appears in the interface, the definition of *T* in the exporter must specify a type with that size and with the standard fundamental operations (the compiler checks this).

Within an exporter, *Defs.T* and *T* conform freely and are assignment compatible. Otherwise, *Defs.T* is treated opaquely there and is *not* equivalent to *T* (except for the purpose of checking exports). You should therefore attempt to write an exporting module entirely in terms of concrete types. Consider the following example:

Interface Module (*Defs*):

```
T: TYPE;
H: TYPE = POINTER TO T;
R: TYPE = RECORD [f: H, ...];
Proc1: PROC [h: H];
Proc2: PROC [r: POINTER TO R];
...
```

Exporting Module:

```
T: PUBLIC TYPE = RECORD [v: ...];
P: TYPE = POINTER TO T;
Proc1: PUBLIC PROC [h: P] = {... h.v ...};
Proc2: PUBLIC PROC [r: POINTER TO Defs.R] = {
  q: P = r.f;
  ... q.v ...};
...
```

If the type of *h* were *Defs.H* in the implementation of *Proc1*, the reference to *h.v* would be illegal. By defining a type such as *P* and using it within the exporter instead of *H*, you can avoid most such problems. (Note that *Proc1* is still exported with the proper type.) This strategy of creating concrete types in one-to-one correspondence to interface types involving *T* fails for record types such as *R* (because of the uniqueness rule for record constructors). In this example, you must use *Defs.R* to define the type of *r* in the implementation of *Proc2*, but a reference to *r.f.v* is illegal. In such cases, a LOOPHOLE-free implementation may require redundant assignments, such as the one to *q*. Alternatively, you should consider making the record type another exported type, and defining its concrete type within the exporter also.

Binding

For each interface containing some exported type *T*, all exporters of that interface must provide equivalent concrete types for *T* (the binder and loader check this). In Mesa 6, the concrete types must in fact be identical; if two modules export *T*, they must obtain the same concrete definition of *T*, e.g., from another shared interface module (typically, a private one).

Control Variables

You can now declare the control variable of a loop as part of the FOR clause attached to that loop. Such an identifier cannot be accessed outside the loop and cannot be updated except by the FOR clause in which it is declared.

Syntax

```
Iteration      ::=     FOR identifier Direction IN LoopRange
               |       FOR identifier : TypeExpression Direction IN LoopRange

Assignation    ::=     FOR identifier _ Expression , Expression
               |       FOR identifier : TypeExpression _ Expression , Expression
```

The forms of Iteration and Assignation with ": TypeExpression" declare a new control variable. That variable cannot be explicitly updated (except by the FOR clause itself). Its scope is the entire LoopStmt introduced by the Iteration or Assignation including any LoopExitsClause. Note, however, that the value of a control variable used in an Iteration is undefined in the FinishedExit.

Extended NIL

In Mesa 6, null values are available for all address-containing types. An address-containing type is one constructed using POINTER, DESCRIPTOR, PROCEDURE, PROGRAM, SIGNAL, ERROR, PROCESS, PORT, ZONE or a LONG or subrange form of one of the preceding. The built-in type STRING is address-containing. A relative pointer or relative descriptor type is not considered to be address-containing in Mesa 6.

Null values are denoted as follows:

If *T* designates any address-containing type, NIL[*T*] denotes the corresponding null value.

Whenever *T* is implied by context, NIL abbreviates NIL[*T*].

If *T* is not implied by context, NIL means NIL[POINTER TO UNSPECIFIED] and thus continues to match any POINTER or LONG POINTER type.

A fault will occur if you attempt to dereference a null value *and* have requested NIL checking; a fault will occur unconditionally if you attempt to transfer control through a null value.

Reject Statement

Within a catch phrase, you can use the statement REJECT to explicitly reject a signal, i.e., to terminate execution of that catch phrase and propagate the signal to the enclosing one. (Note that each catch phrase is currently terminated by an implicit REJECT.)

Process Extensions

Aborting a process now raises the predeclared signal ABORTED. The predeclared types MONITORLOCK and CONDITION are now defined with default initialization. The only client-visible field is *timeout* in CONDITION; its default initial value is 100 ticks.

Restrictions on Assignment

The assignment operations defined upon certain types have been restricted so that variables of those types can be initialized (either explicitly or by default) when they are created but cannot subsequently be updated. A variable is considered to be created at its point of declaration or, for dynamically allocated objects, by the corresponding NEW operation.

In Mesa 6, the following types have restricted assignment operations:

MONITORLOCK

CONDITION

any type constructed using PORT

any type constructed using SEQUENCE

any type constructed using ARRAY in which the component type has a restricted assignment operation.

any type constructed using RECORD in which one of the field types has a restricted assignment operation.

Note that the restrictions upon assignment for a type do not impose restrictions upon assignment to component types. Thus selective updating of fields of a variable may be possible even when the entire variable cannot be updated; e.g., the *timeout* field of a CONDITION variable can be updated by ordinary assignment. Also, you may apply the operator @ to obtain the address of the entire variable in such a situation.

Operational Changes

User Interface

The standard Mesa 6 Compiler reads commands only from the executive's command line; it no longer supports interactive input. During compilation, the display and keyboard are disabled. The cursor provides a limited amount of feedback; it moves down the screen to indicate progress through a sequence of commands and to the right as errors are detected. At the end of compilation, the message "Type Key" is displayed in a flashing cursor if there are errors and you have requested the compiler to pause. Typing `Shift-Swat` aborts the Executive's current command sequence; `Ctrl-Swat` invokes the Mesa Debugger; any other character causes normal exit from the compiler.

A summary of compilation commands is written on the file `Compiler.log` (formerly, `Mesa.typescript`).

Command Line Arguments

The Mesa 6 Compiler allows you to control the association between modules and file names at the time you invoke the compiler. The compiler accepts a series of commands, each of which has the form

$$\texttt{outputFile \_ inputFile[id}_1\texttt{: file}_1\texttt{, ..., id}_n\texttt{: file}_n\texttt{]/switches}$$

Only `inputFile` is mandatory; it names the file containing the source text of the module to be compiled, and its default extension is `.mesa`. Any warning or error messages are written on the file `outputRoot.errlog`, where `outputRoot` is the string obtained by deleting any extension from `outputFile`, if given, otherwise from `inputFile`. If there are no errors or warnings, any existing error log with the same name is deleted at the end of the compilation.

If a list of keyword arguments appears between brackets, each item establishes a correspondence between the name $\texttt{id}_i$ of an included module, as it appears in the DIRECTORY of the source program, and a file with name $\texttt{file}_i$; the default extension for such file names is `.bcd`. (If the name of an included module is not mentioned on the command line, its file name is computed from information in the DIRECTORY statement; see above).

The optional `switches` are a sequence of zero or more letters. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by `-` or ~ to invert the sense of the switch.

If `outputFile` (and _) are omitted, the object code and symbol tables are written on the file `inputRoot.bcd`, where `inputRoot` is `inputFile` with any extension deleted. Otherwise code and symbols are written on `outputFile`, for which a default extension of `.bcd` is supplied. If the compiler detects any errors, the output file is not written and any existing file with the same name is deleted.

The compiler accepts a sequence of one or more commands from the executive's command line (through the file `Com.cm`). Commands are separated by semicolons, but you may omit a semicolon between any two successive identifiers (file names or switches), or between a] and an identifier (but not between an identifier and a `/`). Note that any required semicolon in an Alto Executive command must be quoted.

You can set global switches by a command with an empty file name. In the form `/switches`, each letter designates a different switch. Unless a command to change the global switch settings comes first in the sequence of commands, you must separate it from the preceding command by an explicit semicolon. Note that the form switch/c is no longer available for setting global switches.

Switches

The following compilation options have been added:

Switch	*Option Controlled*
`f`	implementation of floating-point operations
`l`	treatment of long pointers
`y`	warning on runtime calls

If the `f` switch is set, the compiler generates byte code instructions for floating-point operations (these require microcode support); otherwise, it generates calls through the system dispatch vector (SD) to software routines implementing such operations. If the `a` and `l` switches are both set, the compiler generates code using an variant of the Alto Mesa instruction set that implements long pointer accesses to a virtual memory larger than 64K (code generated using the `l` switch cannot be executed on an Alto, even if long pointers are not used). If you specify `-a`, the `l` switch is ignored. The `y` switch indicates that a warning message should be issued whenever the compiler generates code to invoke a runtime procedure (including some "instructions" which are actually implemented in software).

The default settings for these switches are `f`, `-l` and `-y`.

Distribution:
  Mesa Users
  Mesa Group
  SD Support